BOLT BERANEK AND NEWMAN INC

CONSULTING • DEVELOPMENT • RESEARCH

LEVEL II

BBN Report No. 3331

June 1976

INTERLISP PERFORMANCE MEASUREMENTS

Interim Report

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Advanced Research Projects Agency or the U.S. Government.

This research was supported by the Advanced Research Projects Agency under ARPA Contract No. N00014-75-C-1110.

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce for sale to the general public.

BOSTON WASHINGTON CHICAGO HOUSTON LOS ANGELES OXNARD

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 BBN ~~Report No.~~ 3331 ✓	2. GOVT ACCESSION NO. AD-A104	3. RECIPIENT'S CATALOG NUMBER 583
4. TITLE (*and Subtitle*) 6 INTERLISP PERFORMANCE MEASUREMENTS. Interim Report		5. TYPE OF REPORT & PERIOD COVERED 9 Interim Report, 4/1/75 - 6/30/76 1 Apr 75 - 30 Jun 76
		6. PERFORMING ORG. REPORT NUMBER BBN Report No. 3331 ✓
7. AUTHOR(s) 10 Robert Bobrow ~~and~~ Mario Grignetti		8. CONTRACT OR GRANT NUMBER(s) 15 N00014-75-C-1110
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, Mass. 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ✓ ARPA Order ~~No.~~ 3023
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE 11 June ~~1~~976
		13. NUMBER OF PAGES 56
14. MONITORING AGENCY NAME & ADDRESS(*if different from Controlling Office*) 12 64		15. SECURITY CLASS. (*of this report*) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce for sale to the general public.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

LISP, INTERLISP, Memory Usage, Page Faulting, Efficiency, TENEX.

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

This report describes measurements performed for the purpose of determining areas of potential improvement to the efficiency of INTERLISP running under TENEX.

A

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified. LB
SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

060100

BBN Report No. 3331 June 1976

INTERLISP PERFORMANCE MEASUREMENTS

Interim Report

This research was supported by the Advanced Research Projects Agency under ARPA Contract No. N00014-75-C-1110.

INTERLISP PERFORMANCE MEASUREMENTS

Introduction

The advantages of LISP for fast production of large software systems, especially those involving Artificial Intelligence applications, are too well known for us to expound on here. Suffice it to say that systems such as DENDRAL, MACSYMA, SOPHIE, SCHOLAR, LUNAR, etc., could not have evolved and could not have been developed within the time and level of effort they actually required to complete, had it not been for the existence of a sophisticated LISP programming environment, of which perhaps INTERLISP is the best known.

But as all users of these systems know, the blessings of such sophisticated programming environments are not without some serious counterparts. Although programming (debugging, editing, ...) and running finished products written in INTERLISP proceed at a surprisingly fast clip when the total load on the machine is low, degradation of performance increases rapidly - and seemingly non-linearly - to intolerable levels as soon as large numbers of users increase the demands imposed on the computer's resources.

An indication of how bad things can be is provided by the following typical example. In a busy morning, a simple INTERLISP editing command that uses under 300 milliseconds of CPU time takes almost an order of magnitude more elapsed time than would be expected from the average load on the machine. Thus, extremely slow responsiveness in the face of small computational demands is one of the serious problems that we shall investigate.

Another aspect of this performance degradation manifests itself in the behavior of CPU bound jobs, such as compiling. In spite of the TENEX pie-slice scheduler's guaranteed fraction of CPU power, compute bound INTERLISP jobs rarely get more than 50% of their guaranteed CPU power when, again, the load imposed on the machine by other users increases beyond certain limits.

The above cited typical situations provide the motivation and the framework for the work to be described. Our objective was to pin down what aspects of the "INTERLISP cum TENEX" environment were responsible for the observed objectionable behavior, and to propose and implement remedies to improve the situation. More specifically, our goals were to improve system responsiveness for short interactions (e.g. editing) and to increase system efficiency and throughput when executing CPU bound jobs.

To this end we performed an extensive series of measurements covering a variety of aspects of system behavior. We obtained statistics of usage of INTERLISP from different users doing different things; we traced the way INTERLISP uses core, both in the space and the time dimensions; and we identified with high resolution the areas where most instruction executions take place, i.e. where and doing what the system spends most of its time.

Whoever has tried to understand with precision the behavior of a complex system immersed in a time-sharing system knows how difficult it is to actually measure what one wants, and how hard it is to interpret the data one finally obtains. For this reason, we shall endeavor to describe faithfully the methods and procedures used to obtain our data, the conditions under which it was obtained, and our reasons for asserting that it means what we believe it does. Our aim is not only to describe our work and justify our results, but also to make the data and the methodology used to obtain it available to others that may find it useful for their own purposes.

Before embarking in this voyage, however, let us advance here our main conclusions for the benefit of readers not wishing to wade through the rest of the paper.

1) the lack of responsiveness (disproportionately long elapsed times for relatively modest computational demands, or waiting 20 seconds when 3 seconds should have been enough) is due to both the large working set needed by INTERLISP and the particular way the TENEX operating system allows the core-memory allocated to a process to grow to the process' working set size. In order for any significant amount of useful computation to take place in INTERLISP, it is necessary to have from 60 to 100 pages in core; with less than that, program execution is interrupted by page faults at intervals of a millisecond or less. Since TENEX does not do any preloading, and forces a process to grow its working set by page faulting itself up from page 1, it takes roughly 10 seconds (at 100 milliseconds wait time for latency, page management routines, and rescheduling delays) to build up an INTERLISP working

set. At this point (or even sooner) however, since page fault interrupts are considered part of the process' chargeable CPU time, the process would have exceeded its quantum allocation on the high priority interactive queue, and descend to a lower priority scheduling queue. If by the time the second startup of the process occurs (on the lower priority queue), the process is still in the balance set (i.e. has most of its pages in core) some meaningful computation can then begin to take place. If not, the same painful page by page reloading process takes place again.

The remedies to this situation are direct:

a) reduce the size of INTERLISP's working set

b) modify TENEX so that demand paging occurs <u>after</u> initial preloading of the previous working set for the job.

2) Our second main conclusion addresses the issue of basic efficiency. It pertains more definitely to the INTERLISP system itself, and less to TENEX, and has its major impact on compute-bound processes. Briefly, we found that roughly 80% of the instruction fetches occur within 30 pages of shared virtual address space, chiefly the MACRO (or hand-coded) module. In other words, the system spends a majority of its time executing instructions within a relatively small and functionally well-defined area of the INTERLISP address space. It follows that tightening and streamlining code in those sections should bring about the largest payoffs in terms of system operating efficiency. Our measurements

in this regard were of very high resolution. We were able to pin point the most often used 8 word blocks of code in the MACRO area, pointing out in great detail where improvements were needed. Our work on shallow binding, fast function entry, fast type checking and fast CONSing responds to these documented bottlenecks.

Although these are the highlights, there are of course details and subtle complications that will be dealt with more thoroughly in the following sections.

MEASUREMENTS

In order to characterize the computational patterns of INTERLISP running under TENEX, two distinct types of measurements were made. One set of measurements involved the pattern of usage of TENEX resources by several INTERLISP users over an extended period of time. The second set of measurements involved detailed examination of the underlying activity of the INTERLISP system itself as it was performing a number of typical tasks.

Usage patterns and modes of interaction

A very suitable strategy for the amelioration of INTERLISP performance is to concentrate on those patterns of usage that involve a large and perhaps unnecessary amount of computational power, memory, and/or the user's own time. In order to do this, we needed to characterize the actual use of INTERLISP by normal users going about their daily business.

The first set of measurements used the built-in TENEX job parameter statistics (CPU time charged, elapsed time, page faults, time charged within page-fault routines) to monitor the activity patterns of several typical users over an extended period of time.

The data given is for a moderately long run (about 200 events) of editing, compiling and associated debugging operations, typical of much of the activity of the LISP community. An event is defined as a single operation the user requires INTERLISP to do (like a PP or DW command in the editor, or a compilation) which results in a certain amount of CPU time being consumed. The total CPU time required for the run is about 2 minutes.

We plot three quantities. First we split the events up into groups defined by a range of required CPU time for the event (thus one group might be events which took between 200 and 250 milliseconds of CPU time to complete). For each group we then plot the proportion of the total CPU time used by the job which is attributable to events in that group. The graph indicates the chosen CPU time ranges for events. The upper value of the range is given in the left hand column, and the lower value is the previous upper value (all values given in milliseconds). The number of events requiring CPU times withn the range are given in the second column, and the percentage of the total CPU usage attributable to those events is given in the third column. The percentage is also plotted as a bar graph immediately to the right, with scale given below. Note that, if we define interactive events as those that require CPU times of less than 300 milliseconds, only 21% of the total CPU is used in interactive operations. This is actually somewhat higher than we have seen in the uncontrolled data taken from several typical users - this run involves a lot of editing. Thus, the lion's share of the CPU load placed on the system is in long, CPU-bound activities.

The second graph shows the number of events requiring CPU times within each of the chosen ranges. It shows quite graphically that 2/3 (67%) of the total number of events represents "interactive" work.

Thus, any degradation of performance resulting in the increase of elapsed time for events (particularly interactive events) will be very strongly felt by the user - with tremendous frustration judging by typical reactions.

Both of these plots are given in terms of net CPU time - this means that for each interaction we have subtracted the time that TENEX indicates was spent in the page-faulting routines, since that time varies strongly with load. This gives an indication of the "basic time" spent in the various interactions. An indication of the additional CPU time billed because of page-faulting is given in the third graph which gives percentage of interactions by gross CPU time - this includes all TENEX billed time, including page faulting. For this example run at relatively low load, there were 2642 faults, representing approximately 11000 milliseconds of time recorded as spent in the page faulting routines. The overall gross CPU time is 74,800 milliseconds. Thus, about 15% of the billed time is due to page faulting.

The same three graphs are given for an almost identical run at moderate load. While the first two graphs are roughly similar (with the exception of a "spike" at the .5-1.0 second reange due to a number of error recovery (DWIM type) operations caused by execesive mistyping), the third graph shows that there is a notable increase in CPU time billed because of the increase in page-faulting. The total gross CPU time is 132250 milliseconds, with 16053 page faults, accounting for approximately 34,000 milliseconds or over 26% of the billed CPU time. It is interesting to note that there is a difference of about 34 seconds of net cpu time between these two runs as obtained by subtracting the TENEX reported page fault time from the gross CPU time. Since the reported billed time per page fault was over 4 milliseconds during low load, and about 2 milliseconds during high load, it is possible that more time was spent in the page fault routines during the high load situation than was recorded by TENEX. Of course, the slight change in the run accounts for some part of the difference, but probably not more than half. The page faulting behavior is examined in more detail in a later section.

Editing, compiling dwimifying and clispifying - light load

```
AVERAGE LOAD IS: 1.265496
Total net CPU time: 63,550 milliseconds

MAX. CPU | # OF |  PERCENTAGE OF TOTAL USED NET CPU TIME
MILLISEC | EVTS |
         |      |          |    |    |   |    |    |    |    |    |    |
         |      |          |         |        |         |         |
50       | 1    | 0.0      |
100      | 8    | 0.0      |
150      | 23   | 2.0      |**
200      | 34   | 5.0      |*****
250      | 44   | 8.5      |********
300      | 22   | 5.5      |*****
350      | 14   | 4.0      |****
400      | 11   | 3.5      |***
450      | 7    | 2.5      |**
500      | 2    | .5       |
1000     | 4    | 2.5      |**
1500     | 8    | 10.0     |**********
2000     | 2    | 3.0      |***
2500     | 2    | 4.5      |****
3000     | 1    | 2.5      |**
3500     | 2    | 6.5      |******
4000     | 0    | 0.0      |
4500     | 1    | 4.0      |****
5000     | 0    | 0.0      |
10000    | 1    | 5.0      |*****
15000    | 0    | 0.0      |
20000    | 0    | 0.0      |
40000    | 1    | 26.0     |**************************
60000    | 0    | 0.0      |
120000   | 0    | 0.0      |
180000   | 0    | 0.0      |
240000   | 0    | 0.0      |
300000   | 0    | 0.0      |
360000   | 0    | 0.0      |
*INF*    | 0    | 0.0      |
                           |         |         |         |         |
                           |    |    |    |    |    |    |    |    |    |
                           0    5    10   15   20   25   30   35   40   45
```

```
MAX. CPU  | # OF |  PERCENTAGE OF INTERACTIONS OF GIVEN NET CPU TIME
MILLISEC  | EVTS |
          |      |         |    |    |    |    |    |    |    |    |    |
          |      |         |         |         |         |         |
50        | 1    | .5      |
100       | 8    | 4.0     |****
150       | 23   | 12.0    |************
200       | 34   | 18.0    |******************
250       | 44   | 23.0    |***********************
300       | 22   | 11.5    |***********
350       | 14   | 7.0     |*******
400       | 11   | 5.5     |*****
450       | 7    | 3.5     |***
500       | 2    | 1.0     |*
1000      | 4    | 2.0     |**
1500      | 8    | 4.0     |****
2000      | 2    | 1.0     |*
2500      | 2    | 1.0     |*
3000      | 1    | .5      |
3500      | 2    | 1.0     |*
4000      | 0    | 0.0     |
4500      | 1    | .5      |
5000      | 0    | 0.0     |
10000     | 1    | .5      |
15000     | 0    | 0.0     |
20000     | 0    | 0.0     |
40000     | 1    | .5      |
60000     | 0    | 0.0     |
120000    | 0    | 0.0     |
180000    | 0    | 0.0     |
240000    | 0    | 0.0     |
300000    | 0    | 0.0     |
360000    | 0    | 0.0     |
*INF*     | 0    | 0.0     |
                           |         |         |         |         |
                           |    |    |    |    |    |    |    |    |    |
                           0    5    10   15   20   25   30   35   40   45
```

```
Total gross CPU time: 74,800 milliseconds

MAX. CPU ¦ PERCENTAGE OF INTERACTIONS OF GIVEN GROSS CPU TIME
MILLISEC ¦
                ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦
                ¦         ¦         ¦         ¦         ¦
50       ¦ 0.0  ¦
100      ¦ 2.5  ¦**
150      ¦ 10.5 ¦**********
200      ¦ 13.0 ¦*************
250      ¦ 19.5 ¦*******************
300      ¦ 13.0 ¦*************
350      ¦ 11.0 ¦***********
400      ¦ 6.0  ¦******
450      ¦ 4.0  ¦****
500      ¦ 3.0  ¦***
1000     ¦ 5.0  ¦*****
1500     ¦ 2.0  ¦**
2000     ¦ 2.5  ¦**
2500     ¦ 1.0  ¦*
3000     ¦ 1.0  ¦*
3500     ¦ .5   ¦
4000     ¦ 1.0  ¦*
4500     ¦ 0.0  ¦
5000     ¦ .5   ¦
10000    ¦ .5   ¦
15000    ¦ 0.0  ¦
20000    ¦ 0.0  ¦
40000    ¦ .5   ¦
60000    ¦ 0.0  ¦
120000   ¦ 0.0  ¦
180000   ¦ 0.0  ¦
240000   ¦ 0.0  ¦
300000   ¦ 0.0  ¦
360000   ¦ 0.0  ¦
*INF*    ¦ 0.0  ¦
                ¦         ¦         ¦         ¦         ¦
                ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦
                     0    5    10   15   20   25   30   35   40   45
```

Editing, compiling, dwimifying and clispifying - moderately heavy load

```
AVERAGE LOAD IS: 6.637456
Total net CPU time: 98,250 milliseconds

MAX. CPU | # OF |  PERCENTAGE OF TOTAL USED NET CPU TIME
MILLISEC | EVTS |
         |      |
         |      |      |    |    |    |    |    |    |    |    |    |
         |      |                |         |         |         |
50       | 1    | 0.0  |
100      | 10   | 0.0  |
150      | 23   | 1.5  |*
200      | 31   | 3.5  |***
250      | 26   | 4.0  |****
300      | 19   | 3.5  |***
350      | 16   | 3.5  |***
400      | 7    | 1.5  |*
450      | 4    | 1.0  |*
500      | 2    | .5   |
1000     | 24   | 11.5 |***********
1500     | 7    | 6.5  |******
2000     | 4    | 5.0  |*****
2500     | 2    | 3.5  |***
3000     | 1    | 2.0  |**
3500     | 2    | 5.0  |*****
4000     | 1    | 3.0  |***
4500     | 0    | 0.0  |
5000     | 1    | 3.5  |***
10000    | 1    | 6.5  |******
15000    | 1    | 8.0  |********
20000    | 0    | 0.0  |
40000    | 1    | 20.5 |********************
60000    | 0    | 0.0  |
120000   | 0    | 0.0  |
180000   | 0    | 0.0  |
240000   | 0    | 0.0  |
300000   | 0    | 0.0  |
360000   | 0    | 0.0  |
*INF*    | 0    | 0.0  |
                       |         |         |         |         |
                       |    |    |    |    |    |    |    |    |    |
                       0    5    10   15   20   25   30   35   40   45
```

```
MAX. CPU | # OF | PERCENTAGE OF INTERACTIONS OF GIVEN NET CPU TIME
MILLISEC | EVTS |
         |      |       |    |    |   |    |    |    |    |    |    |
         |      |       |         |         |         |         |
50       | 1    | .5    |
100      | 10   | 5.0   |*****
150      | 23   | 12.5  |************
200      | 31   | 16.5  |****************
250      | 26   | 14.0  |**************
300      | 19   | 10.0  |**********
350      | 16   | 8.5   |********
400      | 7    | 3.5   |***
450      | 4    | 2.0   |**
500      | 2    | 1.0   |*
1000     | 24   | 13.0  |*************
1500     | 7    | 3.5   |***
2000     | 4    | 2.0   |**
2500     | 2    | 1.0   |*
3000     | 1    | .5    |
3500     | 2    | 1.0   |*
4000     | 1    | .5    |
4500     | 0    | 0.0   |
5000     | 1    | .5    |
10000    | 1    | .5    |
15000    | 1    | .5    |
20000    | 0    | 0.0   |
40000    | 1    | .5    |
60000    | 0    | 0.0   |
120000   | 0    | 0.0   |
180000   | 0    | 0.0   |
240000   | 0    | 0.0   |
300000   | 0    | 0.0   |
360000   | 0    | 0.0   |
*INF*    | 0    | 0.0   |
                        |         |         |         |         |
                        |    |    |    |    |    |    |    |    |    |
                        0    5   10   15   20   25   30   35   40   45
```

```
Total gross CPU time: 132,250 milliseconds

MAX. CPU | PERCENTAGE OF INTERACTIONS OF GIVEN GROSS CPU TIME
MILLISEC |
                |    |    |    |    |    |    |    |    |    |
                |         |         |         |         |

50       | 0.0  |
100      | 1.5  |*
150      | 7.5  |*******
200      | 8.0  |********
250      | 10.5 |**********
300      | 11.0 |***********
350      | 7.5  |*******
400      | 8.5  |********
450      | 4.0  |****
500      | 5.5  |*****
1000     | 19.0 |*******************
1500     | 3.5  |***
2000     | 3.0  |***
2500     | 2.0  |**
3000     | 0.0  |
3500     | 1.0  |*
4000     | 1.0  |*
4500     | .5   |
5000     | .5   |
10000    | .5   |
15000    | 1.0  |*
20000    | 0.0  |
40000    | .5   |
60000    | 0.0  |
120000   | 0.0  |
180000   | 0.0  |
240000   | 0.0  |
300000   | 0.0  |
360000   | 0.0  |
*INF*    | 0.0  |
                |         |         |         |         |
                |    |    |    |    |    |    |    |    |    |
                        0    5   10   15   20   25   30   35   40   45
```

Change in Interfault interval between low and moderate loads

As described above, one of our observations is that the major non-linear effect of machine load occurs as a result of vastly increased page faulting, particularly for short, supposedly "interactive" jobs. We plot the average time (net CPU - not counting time TENEX indicates to be "page fault time") between page faults for different net CPU length interactions. This data is plotted for the two runs of the typical editing, compiling, etc. job described above, at two different load averages. Note that we only have control over load average, we do not have any direct measurement of actual memory contention. The low load average run is at a value slightly higher than the "dead of night load", but roughly comparable. The high load average run is only normal moderate afternoon loading - that is already bad enough in terms of page faulting, so that really horrible load averages such as the 10-20 range are not shown (and heaven forbid the 20-30 load range).

Low load average - about 1.2

2642 total page faults

```
MAX. CPU  | # OF |  INTERFAULT INTERVAL - MILLISECONDS NET CPU TIME
MILLISEC  | EVTS |
          |      |         |    |    |    |    |    |    |    |    |    |
          |      |         |         |         |         |         |
50        | 1    | 0.0     |
100       | 8    | 15.5    |***************
150       | 23   | 33.5    |*********************************
200       | 34   | 43.0    |*******************************************
250       | 44   | 62.0    |**********************************************+
300       | 22   | 71.0    |**********************************************+
350       | 14   | 85.0    |**********************************************+
400       | 11   | 32.0    |********************************
450       | 7    | 23.5    |***********************
500       | 2    | 19.5    |*******************
1000      | 4    | 8.0     |********
1500      | 8    | 13.0    |*************
2000      | 2    | 12.5    |************
2500      | 2    | 6.0     |******
3000      | 1    | 19.0    |*******************
3500      | 2    | 38.0    |**************************************
4000      | 0    | 0.0     |
4500      | 1    | 17.0    |*****************
5000      | 0    | 0.0     |
10000     | 1    | 20.0    |********************
15000     | 0    | 0.0     |
20000     | 0    | 0.0     |
40000     | 1    | 13.0    |*************
60000     | 0    | 0.0     |
120000    | 0    | 0.0     |
180000    | 0    | 0.0     |
240000    | 0    | 0.0     |
300000    | 0    | 0.0     |
360000    | 0    | 0.0     |
*INF*     | 0    | 0.0     |
                           |         |         |         |         |
                           |    |    |    |    |    |    |    |    |    |
                           0    5    10   15   20   25   30   35   40   45
```

<u>Moderate load average - about 6.6</u>

16053 total page faults

```
MAX. CPU  | # OF |  INTERFAULT INTERVAL - MILLISECONDS NET CPU TIME
MILLISEC  | EVTS |
          |      |        |    |    |    |    |    |    |    |    |    |
          |      |        |         |         |         |         |
50        | 1    | 0.0    |
100       | 10   | 3.0    |***
150       | 23   | 7.0    |*******
200       | 31   | 6.0    |******
250       | 26   | 13.5   |*************
300       | 19   | 6.0    |******
350       | 16   | 2.5    |**
400       | 7    | 5.0    |*****
450       | 4    | 6.0    |******
500       | 2    | 5.5    |*****
1000      | 24   | 5.0    |*****
1500      | 7    | 3.5    |***
2000      | 4    | 4.0    |****
2500      | 2    | 2.5    |**
3000      | 1    | 4.0    |****
3500      | 2    | 4.5    |****
4000      | 1    | 4.5    |****
4500      | 0    | 0.0    |
5000      | 1    | 10.0   |**********
10000     | 1    | 6.0    |******
15000     | 1    | 4.5    |****
20000     | 0    | 0.0    |
40000     | 1    | 13.0   |*************
60000     | 0    | 0.0    |
120000    | 0    | 0.0    |
180000    | 0    | 0.0    |
240000    | 0    | 0.0    |
300000    | 0    | 0.0    |
360000    | 0    | 0.0    |
*INF*     | 0    | 0.0    |
                          |         |         |         |         |
                          |    |    |    |    |    |    |    |    |    |
                          0    5    10   15   20   25   30   35   40   45
```

The measurements just described provide a picture of what "TENEX believes" are the characteristics of LISP execution in a time-shared environment. We use the expression "TENEX believes" because, as in any time-sharing environment, the usage parameters shown for a given job depend heavily on the system load over the course of the job. In part this is due to the necessarily approximate allocation of system overhead among the active jobs, which appears as an addition to the computational resources the jobs would consume if they were running alone. More important, however, is the fact that both the actual amount of overhead and the allocation of this overhead to different jobs varies substantially with different job mixes. A job with given memory requirements for example, will page-fault much more often when it is competing for core space with other memory-hungry jobs (or many small-memory jobs) than when it is running in less memory-competitive environments. Handling these page faults results in additional overhead (CPU time) charged to the job.

Excessive page-faulting causes a dramatic lengthening of the elapsed time for a job not only because disk latency increases the effective cycle time for memory references but because, more importantly perhaps, such behavior can interact with the scheduler, resulting in a job with basically interactive CPU requirements (a small fraction of a second of CPU time needed between interactions with the user) being dropped from the high-priority interactive queue and placed on the less-frequently serviced compute-bound queues. I/O contention causes similar problems in increasing overhead and wait times for jobs competing for use of shared devices such as the disk. Thus, for no fault that is intrinsically their own, certain jobs may be penalized because their overhead-burdened CPU consumption makes the scheduler decide that they belong in a lower-priority queue. In situations of high memory contention this effect can pyramid, because during the wait on the low priority queue the job may have most of its in-core pages removed from core, and thus have to fault many more times than it would have had to if it were allowed to finish its short CPU interaction.

In short, the usage parameters vary because the memory load and CPU demand on the system change with different mixes of jobs, and these load factors strongly affect the interaction of a user program (e.g. INTERLISP) and the TENEX memory manager, i/o drivers and scheduler.

Memory and CPU usage of INTERLISP as a separate system

The TENEX statistics correlated well with what the "monitored" users experienced (and thus what the "typical user" would be likely to experience) in operating INTERLISP under TENEX. While these statistics suggested several changes to the TENEX system, they were insufficient to provide a guide to the modifications to INTERLISP which would most improve the operation of the combined INTERLISP/TENEX system. This was due both to the coarseness of the measurements with regard to the operation of INTERLISP itself as an independent job, as well as to the great difficulty of characterizing the details of the actual interaction between the two systems (or even characterizing the system load parameters which prevailed during the measurements). Thus it was necessary to obtain an entirely independent characterization of the memory and CPU usage of INTERLISP in executing typical operations.

This independent characterization consisted of a series of related measurements based on a PDP-10 simulator program running under TENEX. The simulator is a program which sits in a user's address space, and essentially single-steps through a user program. The simulator takes over from the PDP-10 hardware the job of computing the effective addresses for each of the user program's instructions, and provides hooks to allow a measurement program to record the memory reference pattern of the user job in any degree of detail desired. It is important to note that the simulator sees a JSYS monitor call as _one instruction_ - NO ANALYSIS IS MADE OF TIME SPENT IN THE TENEX MONITOR DOING I/O AT THE USER'S BEHEST. Thus, any program involving i/o will seem to execute fewer instructions (as counted by the simulator) than are actually executed when the program itself is run on the PDP-10. There are many other subtleties involved in understanding precisely how the simulator works and how the data was analyzed. However, we feel that these details are best discussed after we have presented the gist of the measurement results.

Page Faulting versus Allowed Working Set

INTERLISP has acquired a reputation as a "core hog" - a program that requires huge amounts of core in order to run. One of the most interesting things to do with the page reference data is to determine exactly how much core INTERLISP needs to run. Of course this is a poorly defined question - what is interesting is the tradeoff between the expected number of page faults (or the expected time between page faults) and the number of pages allowed in the working set. It is difficult to determine the tradeoff mentioned above in the case of TENEX, because the page management algorithms in TENEX are rather complicated and are influenced by the existence of pages shared among several processes (which may cause TENEX to lose track of the last time a given process used a shared page). Thus, we have resorted to using the page usage data in conjunction with a simplified page management model in order to give some indication of the effect of working set size on page fault rate.

We have produced graphs showing the number of page faults expected for several measured programs for allowed working set sizes ranging from about 40 to 200 pages, using an approximation to a simple page management algorithm. The assumed page management routine is a simple LEAST RECENTLY USED (LRU) algorithm working with a fixed size working set. Thus, when a process starts up it begins to fault in pages, until it has brought in as many pages as there are allowed in the particular fixed size of the working set. The next time that a page not in the working set is referenced, the page in the working set least recently referenced is removed from the working set and replaced with the new page. The same process goes on for each page referenced which is not in the current working set. It is possible to simulate the behavior of such a page management algorithm for different fixed size working sets and to determine the number of page faults that would result for a given process for which we have page reference data.

We present below the graphs of page faults versus allowed fixed working set size for three typical program executions, and include tabular data for other measured programs in the appendix. A number of inferences can be drawn from them, depending on various assumptions that might be made about paging behavior on TENEX and on the parameters of interest.

The first example, referred to as DWCL, involves three typical user operations invoked under the LISP editor - "dwimifying" an expression, "clispifying" an expression, and PRETTYPRINTing the expression. (In the appendix we present the data for a much longer run, called EDIT/CLEANUP, involving many editing operations of substitution, structure changing, etc., obtained by repeating a protocol of an actual large scale debugging session using the simulator.) The second example, referred to as REGCOM, involves the compilation of a set of functions which are already in core (i.e. COMPILE as against TCOMPL, so no file reading operations are included). The third example is the operation of the structure generator from the DENDRAL program, generating the possible structures of the compound C_4H_6 (it is referred to as CONGENSIM - the CONGEN simulation).

The graphs of page-faulting behavior for these examples are given below. The first column (labelled "WORKING SET SIZE") gives the number of pages allowed to accumulate in core before the LRU algorithm is used to replace old pages with new ones (causing page faults). The second column (labelled "PAGE FAULTS") is the number of page replacements that occur for the corresponding working set size. These two columns give a complete tabular representation of the data. The data is graphed to the right of the tabular representation, with the Y-axis being allowed working set size (as given in the first column), and the X-axis being the number of page faults per 20000 memory references (this serves to make the graphs of different runs more comparable), with the scale for the number of page faults being given below the graph. As is indicated, there are approximately 1.2 million memory references which took place in the course of the dwimification, clispification and PRETTYPRINTing. Note that the number of instructions executed in monitor mode (for the i/o in PRETTYPRINTing) are not accounted for, nor are any instructions executed in TENEX for page management, scheduling, etc.

Example: DWCL

1204224 Memory references in example

```
Allowed ¦ Page     :          Page faults per 20000 memory references
Working ¦ Faults   :
Set     ¦          :
        ¦          :
   248  ¦   240    :       *
   240  ¦   240    :       *
   232  ¦    241   :       *
   224  ¦    245   :       *
   216  ¦    248   :       *
   208  ¦    253   :       *
   200  ¦    262   :        *
   192  ¦    286   :         *
   184  ¦    310   :         *
   176  ¦    318   :          *
   168  ¦    335   :          *
   160  ¦    347   :           *
   152  ¦    372   :           *
   144  ¦    397   :            *
   136  ¦    416   :             *
   128  ¦    469   :               *
   120  ¦    574   :                  *
   112  ¦    616   :                    *
   104  ¦    700   :                      *
    96  ¦    825   :                           *
    88  ¦    985   :                                *
    80  ¦   1243   :                                        *
    72  ¦   1551   :                                                  *
    64  ¦   1916   :                                                  *
    56  ¦   2535   :                                                  *
    48  ¦   3867   :                                                  *
    40  ¦   8586   :                                                  *
                   ....................................................
                   :0         5        10        15        20       >>20
```

```
Example: REGCOM

2037760 Memory references in example


Allowed ¦ Page     :          Page faults per 20000 memory references
Working ¦ Faults   :
Set     ¦          :
        ¦          :
   240  ¦   236    :    *
   232  ¦   237    :    *
   224  ¦   238    :    *
   216  ¦   242    :    *
   208  ¦   249    :    *
   200  ¦   277    :     *
   192  ¦   289    :     *
   184  ¦   292    :     *
   176  ¦   303    :     *
   168  ¦   308    :     *
   160  ¦   320    :     *
   152  ¦   342    :      *
   144  ¦   372    :      *
   136  ¦   428    :        *
   128  ¦   520    :         *
   120  ¦   797    :               *
   112  ¦   895    :                 *
   104  ¦   962    :                  *
    96  ¦  1041    :                    *
    88  ¦  1128    :                      *
    80  ¦  1251    :                        *
    72  ¦  1420    :                           *
    64  ¦  1689    :                                 *
    56  ¦  2168    :                                          *
    48  ¦  3020    :                                                  *
    40  ¦  5236    :                                                  *
                   ....................................................
                   :0         5        10        15        20       >>20
```

Example: CONGENSIM

2283520 Memory references in example

```
Allowed ¦ Page      :          Page faults per 20000 memory references
Working ¦ Faults    :
Set     ¦           :
        ¦           :
  200   ¦    193    :  *
  192   ¦    196    :  *
  184   ¦    197    :  *
  176   ¦    204    :   *
  168   ¦    212    :   *
  160   ¦    219    :   *
  152   ¦    225    :   *
  144   ¦    235    :   *
  136   ¦    255    :   *
  128   ¦    312    :    *
  120   ¦    400    :      *
  112   ¦    480    :       *
  104   ¦    570    :         *
   96   ¦    679    :           *
   88   ¦    817    :             *
   80   ¦   1016    :                 *
   72   ¦   1217    :                    *
   64   ¦   1593    :                           *
   56   ¦   2045    :                                   *
   48   ¦   2936    :                                                  *
   40   ¦   5401    :                                                  *
                    ....................................................
                    :0         5        10        15        20       >>20
```

Interpretation of page faulting results

There are a number of subtleties that should be borne in mind in looking at the data. In the first place, the number of page faults is given assuming that the job starts from scratch, with no pages in core. Once the job is running, it is able to keep its entire allowed working set with no losses, throughout the entire run, simply bringing in new pages and swapping out LRU pages. If one wishes to make an estimate of the frequency of page faults "in the steady state" for a compute bound job, one should probably assume that the job has its full working set in, and count faults after that. Thus, for this purpose, one should subtract the size of the allowed working set from the fault count for the given working set, in order to determine how many faults occurred in the steady state condition. However, if one is considering the number of faults likely to occur if the interaction starting the example occurs several seconds after the last user interaction, then the number of page faults as stated are meaningful under the standard TENEX page management operation - by the end of a few seconds of waiting for the user to initiate an interaction, the user's program is probably no longer in core because of competition with other jobs demanding memory in order to run.

The substantial flurry of page faults needed to start up an interaction when the program is not in core might account for the difference in responsiveness felt between night-time and daytime running of INTERLISP - at night there are times when the number of users is small enough that the core allocation for a user does not decay for quite a while - conceivably two or three LISPs could reside in core and not be swapped out while waiting for user's responses. Thus, the response to a request (similar to previous requests in terms of the particular pages needed to execute the request) can occur immediately, with relatively little page faulting. During heavier memory contention times, the same request may require over a hundred page faults just to initialize the working set. In turn, the charge for this faulting may take the job off the interactive queue and thus cause a delay until the job starts up on the lower queue - during which time the pages used by the job can start to trickle out due to contention by other jobs.

Assuming that the jobs in question have a high enough priority to run to completion without being removed from core, one can ask how the billed CPU time for the job varies as a function of the allowed working set size. Assuming that TENEX charges an average of about 3 milliseconds of CPU time per page fault, a page faulting rate of one fault per 3 milliseconds would double the charged time for the job. By comparing the number of memory references reported by the simulator to the billed CPU time for a given job (subtracting off time TENEX attributes to paging) we find that each memory reference accounts for about 1.5 microseconds of CPU time (this includes memory reference time, pager time, and the time to execute instructions) on the average. Thus, the billed time doubles when there is one page fault every 2000 memory references. In the editing run this corresponds to a working set size of approximately 115 pages, for the compilation example to about 100 pages, and for the CONGENSIM example to about 76 pages. (These figures are based on the total number of page faults given by the simulator as plotted above. For the "steady state", corresponding sizes are about 100, 70, and 68).

Another interesting question is how the potential elapsed time for a job varies depending on the working set. If one assumes that the minimum time it takes to fetch a page from disc is about one disc latency plus the TENEX billed CPU time per fault, one can say that the minimum elapsed time for a faulted reference is about 30 milliseconds, corresponding to about 20000 regular memory references. Thus, a page fault rate of one fault per 20000 references would cause a doubling of potential elapsed time. Other estimates of effective elapsed time per fault can be made, to take into account scheduling overhead and waits, etc. These estimates range up to 100 milliseconds per fault. This would correspond to 65000 references. There is also a question as to what constitutes an acceptable increase in elapsed time. On the pie slice scheduler, if the user has a 10% slice, then a multiplication of elapsed time by 10 (due to waits for faults or due to scheduling) is not unreasonable. This corresponds to somewhere between 2000 and 6500 references per page fault, depending on estimates of elapsed time to resolve a fault.

Composition of a Working Set

Given that one intends to reduce the working set of INTERLISP in order to reduce page faulting, the question arises as to what the working set of a typical program is made up of. Since the concept of LISP is associated with the notion of list-structure and the existence of large data bases of list structure, one might expect that much of the working space is tied up in list structure. Given this, one might try to reduce the working set by such techniques as linearization and compactification of list structure. In fact, for the programs measured, lists take only a relatively small amount of the working space relative to other items.

Taking the page reference data, we simulated an LRU algorithm for four sizes of working set - 75 pages (a rather cramped set), 100 pages (still small), 125 (reasonable), and 150 pages (a fairly generous one). At intervals we determined which pages were in the working set and what their data type was. We distinguished among several different types of data -

MACRO - hand code part of system
COMPILED CODE - array space with instruction fetch references
ARRAYS - array space with no instruction fetches
STACKS - control and variable binding stacks
LISTS - CONS cell area
ATOMHT - hash table for atoms
ATOMS - atom header area
PNAME - print names of atoms
STRING - characters in strings
STRING POINTERS - pointers to bounds of individual strings
FIXED NUMBERS - fixed point numbers
OTHER - stack pointers, etc.

The results for single programs seemed fairly stable in time, and reasonably consistent from one program to another. We have plotted the composition of the working set for several programs, and include complete tabular data here. The data given averages the composition of the working set over the course of each program's execution, the time-varying data are available, but do not seem to be of any greater interest than the averaged data.

COMPILEMEASURE

MACRO	CCODE	ARRAY	STACK	LISTS	ATOMHT	ATOMS	FPNUM	FXNUM	STRPT	PNAME	STR	OTHER	PAGES
20	22	3	3	11	0	11	0	2	0	2	0	1	75
21	35	4	3	16	0	13	0	2	0	4	0	2	100
23	46	5	4	21	0	14	0	2	2	5	2	1	125
22	58	6	3	29	2	15	0	2	2	6	2	2	150

```
111111111111¦2222222222222222¦333333¦444444¦6¦9S          75 pages

111111111¦2222222222222222222¦3333333¦44444¦6¦9S          100 pages

11111111¦22222222222222222222¦3333333¦4444¦6¦79S          125 pages

111111¦222222222222222222222¦333333333¦44445679S          150 pages

       ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦
       1    2    3    4    5    6    7    8    9    1          percentage
       0    0    0    0    0    0    0    0    0    0          of working
                                                    0          set
```

<u>Legend - numbers signify data types:</u>

1 = MACRO code
2 = COMPILED code and ARRAY
3 = LISTS
4 = ATOMS
5 = ATOMHT
6 = PNAMES
7 = STRINGS
8 = STRPTRS
9 = FIXNUMS
S = STACK

CONGENSIM

MACRO	CCODE	ARRAY	STACK	LISTS	ATOMHT	ATOMS	FPNUM	FXNUM	STRPT	PNAME	STR	OTHER	PAGES
14	31	1	4	7	0	14	0	1	0	1	0	2	75
16	45	1	5	11	1	18	0	1	0	1	0	2	100
16	58	3	5	15	1	21	0	2	0	2	0	2	125
21	66	1	5	18	5	24	0	3	0	5	0	2	150
21	77	4	5	21	9	25	0	4	0	7	0	2	175

```
111111111¦22222222222222222222¦3333¦44444444469SS         75 pages

1111111¦2222222222222222222222¦3333¦444444445 69S         100 pages

111111¦22222222222222222222222¦33333¦44444444 69S        125 pages

111111¦2222222222222222222222¦33333¦4444444¦5¦69S         150 pages

    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦
    1    2    3    4    5    6    7    8    9    1         percentage
    0    0    0    0    0    0    0    0    0    0         of working
                                                 0         set
```

<u>Legend - numbers signify data types:</u>

1 = MACRO code
2 = COMPILED code and ARRAY
3 = LISTS
4 = ATOMS
5 = ATOMHT
6 = PNAMES
7 = STRINGS
8 = STRPTRS
9 = FIXNUMS
S = STACK

PARSEMEASURE

MACRO	CCODE	ARRAY	STACK	LISTS	ATOMHT	ATOMS	FPNUM	FXNUM	STRPT	PNAME	STR	OTHER	PAGES
16	19	2	3	12	1	18	0	0	0	2	0	2	75
18	29	3	3	18	3	20	0	0	0	3	0	2	100
19	40	4	3	24	5	22	0	0	0	5	0	3	125
20	50	4	3	29	9	24	0	0	0	8	0	3	150
21	59	7	3	31	11	28	0	2	0	9	0	4	175

```
1111111111¦2222222222222¦3333333¦44444444444¦56SS          75 pages

11111111¦222222222222222¦33333333¦444444444¦5¦6SS         100 pages

1111111¦22222222222222222¦333333333¦44444444¦5¦6¦S        125 pages

11111122222222222222222¦333333333¦4444444¦55¦66¦S         150 pages

        ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦    ¦
        1    2    3    4    5    6    7    8    9    1      percentage
        0    0    0    0    0    0    0    0    0    0      of working
                                                     0      set
```

<u>Legend - numbers signify data types:</u>

1 = MACRO code
2 = COMPILED code and ARRAY
3 = LISTS
4 = ATOMS
5 = ATOMHT
6 = PNAMES
7 = STRINGS
8 = STRPTRS
9 = FIXNUMS
S = STACK

TASK1NLS

MACRO	CCODE	ARRAY	STACK	LISTS	ATOMHT	ATOMS	FPNUM	FXNUM	STRPT	PNAME	STR	OTHER	PAGES
19	18	2	4	4	5	11	0	1	2	5	2	2	75
21	27	3	5	7	8	16	0	1	2	8	3	2	100
22	35	2	5	10	11	19	0	1	3	9	4	4	125
22	43	5	5	12	14	23	0	2	3	11	5	5	150
23	55	6	3	19	14	25	0	4	4	14	4	4	175

```
111111111111|222222222222|3|4444444|55|6667|89SS          75 pages

1111111111|22222222222222|33|4444444|555|666|7|89S        100 pages

11111111|22222222222222|333|4444444|5555|666|789SS        125 pages

1111111|22222222222222|333|4444444|5555|666|7|89S         150 pages

       |    |    |    |    |    |    |    |    |    |
       |    |    |    |    |    |    |    |    |    |
       1    2    3    4    5    6    7    8    9    1          percentage
       0    0    0    0    0    0    0    0    0    0          of working
                                                    0          set
```

<u>Legend - numbers signify data types:</u>

1 =	MACRO code
2 =	COMPILED code and ARRAY
3 =	LISTS
4 =	ATOMS
5 =	ATOMHT
6 =	PNAMES
7 =	STRINGS
8 =	STRPTRS
9 =	FIXNUMS
S =	STACK

On the average, over half of the working set is taken up with program. The MACRO code seems to be referenced quite often, as indicated by the fact that all the MACRO code needed by a program seems to be in the working set for 100 pages, and no extra MACRO code comes in at 150 pages. Thus, as you go from 100 to 150 pages the "execution code" that is added is almost entirely compiled LISP. Note also that atoms and their ancillary storage are heavily referenced - adding together ATOMHT, ATOMS and PNAMES one gets over 20% of the working set. The remaining 25% is divided up among the other items, with list structure taking only 10-15% of the space.

This data suggests that the three best places to look to reduce working set size are MACRO code, compiled code and atoms. Other data reported below indicate that while 20 pages of the MACRO code are referenced, fewer than 5000 words (10 pages) of the MACRO code are actually used in running the given examples (e.g. the MACRO code used for error recovery, backtracing, etc. are not being used, but they are intertwined with the other code). Thus, by reorganizing the MACRO code about 10 pages can be saved. It is possible that good reorganization can do even better by taking into account the statistical patterns of references within the MACRO code to group together instructions commonly used together. Because the compiled code is the largest single data type, it is reasonable to spend time looking to improve the compiler to produce more compact code. A 10% reduction in size of the compiled code could reduce the working set by 3 to 5 pages. Finally, the large amount of space used by atoms and their ancillary data suggests that compactification of atoms might be useful. In the current system each atom requires four words of storage plus its PNAME - in order to allow it to have a top level value, a property list and a function call. The hash table entry is one word, and the atom header takes three words, since it must hold a full-word function cell, the PNAME pointer, the property list pointer and the value pointer. Other data we have collected suggest that this is quite wasteful, that few atoms have all three features, and that many atoms are used entirely as "indicators" and have only their PNAME and no property list, value or function definition. It is conceivable that this might be taken into account in designing a new structure for atoms.

Counts of references to various page types

One way of determining the general pattern of activity of INTERLISP is to find the actual number of references to a certain type of page during the run of a program. We collected this data and obtained two surprising results - even for large compiled programs, over 80% of the instructions executed were actually ones in the hand-coded part of the LISP kernel; although LISP is associated with the concept of list-processing, fewer than 1.7% of all memory references (instruction fetch and data read or write) go to list structure space. We give the figures for several example programs below. The numbers refer to the fraction of the total number of memory references made by the given program to the particular type of page. "R/W" signifies read/write references to the page, "Instruction fetch" indicates references to memory to obtain instructions. The page types are indicated as follows:

MACRO: instruction portion of hand coded assembly language kernel
CCDAR: compiled code and/or arrays
ASC&V: constants and temporary storage associated with MACRO
PSTAK: the variable binding PDL
CSTAK: the control PDL
LISTS: CONS cell pages
ATOMS: atom header pages
PNAME: pages containing the print name character strings for atoms
NUMS: fixed and floating point numbers
PAGE0: the accumulators (registers) and the UUO trap location

	PARSEMEASURE	CONGENSIM	COMPILEMEASURE	EDIT/CLEANUP	SUBNET
Total Instruction fetch:	.563	.565	.570	.569	.482
Total R/W:	.437	.435	.430	.431	.518
MACRO: Instruction fetch	.514	.452	.438	.444	.362
CCDAR: Instruction fetch	.040	.095	.121	.113	.108
MACRO: R/W	.017	.032	.025	.030	.108
CCDAR: R/W	.009	.024	.022	.023	.041
ASC&V: R/W	.058	.096	.074	.094	.068
PSTAK: R/W	.110	.067	.097	.066	.076
CSTAK: R/W	.090	.117	.095	.106	.082
LISTS: R/W	.012	.013	.015	.010	.016
ATOMS: R/W	.022	.003	.003	.004	.003
ATOMHT: R/W	.000	.000	.000	.001	.000
PNAME: R/W	.001	.000	.001	.004	.001
NUMS: R/W	.000	.000	.001	.000	.000
PAGE0: Instruction fetch	.009	.018	.011	.013	.012
PAGE0: R/W (registers, UUO word)	.118	.083	.097	.092	.123

Detailed instruction fetch measurements on MACRO code - bottlenecks

The second set of measurements was made to determine exactly where the CPU time used in performing typical INTERLISP tasks is spent. Given that the vast majority of the INTERLISP system consists of compiled LISP code rather than hand-coded assembly language (about 200k words of compiled LISP code and about 15k words of hand-written MACRO code) one might expect that a substantial portion of the computation done by LISP consists of executing compiled code. This is reinforced by the fact that over half of the memory required in the working set for a given program is in the compiled code. However, as revealed by the instruction fetch data above, approximately 80% of the instructions being executed were part of the hand-coded kernel of the INTERLISP system the MACRO code. Thus, we decided to take a more detailed look at the distribution of instruction fetches in the hand-coded kernel.

The simulator was modified to record in detail the pattern of instruction fetches that occurred within the macro code. All memory references outside the range occupied by the hand-code and its temporary data storage were lumped together. Within the hand-code area fetch and read/write counts were kept for contiguous 8-word chunks of memory. While it would have been somewhat more meaningful to record data in terms of functional components of the hand-code (e.g. particular subroutines), the table that would have been required was to large, and the time overhead prohibitive. The use of 8-word chunks allowed us to localize references sufficiently to determine the functional chunks by after-measurement analysis.

The resulting data produced a rather strong, and to some people a surprising result. If the 8 word chunks were ordered (for each program) by the number of fetches made within that chunk, then for all programs measured the top 30 chunks accounted for over 48% of the total instruction fetches made by the program. In fact, the average over 12 quite different types of programs was that over 60% of the instruction fetches for a program were contained within the program's top 30 chunks.

It was not only the case that each program had its own "top 30" chunks - the union of the sets of "top 30" chunks had only 54 distinct chunks! Moreover, 45 chunks covered over 50% of the references made by all of the programs. Thus, fewer than 350 words of hand-code (possibly fewer than 300 words since many of the chunks contained obviously low-probability code) accounted for the lion's share of the execution time taken by INTERLISP.

On the basis of this data we were able to pinpoint a small number of high-priority portions of the hand-code to optimize. As it turned out, there was extremely high agreement between the data and the "educated guesses" of the knowledgeable members of the INTERLISP community - the worst offenders had been predicted ahead of time by many of the people familiar with the implementation, and there were almost no qualitative surprises - only the sheer concentration of the instruction fetches was surprising.

While the exact core location and time spent are useful to the systems programmers in determining what words of the MACRO code should be carefully tightened, this level of detail seems unnecessary for this report. Thus, we will give primarily the highlights of the results. The gory details will be made available to those who request them.

The single largest bottleneck in the system turned out to be the procedure for looking up variable bindings on the stack. This took up between 10% and 45% of the total instructions executed, with an "average" (weighted equally over all measured programs) of over 20%. Programs which were block compiled tended to have the lower values of time spent in variable lookup, but still substantial amounts. The next greatest amount of time, averaging 9%, of the instruction fetches, lay in the function calling sequence, followed by about 8% of instruction fetches in the type checking routines. If the time spent in the UUO word and UUO dispatcher are added to these times, the total time spent in the function call and type checking bottleneck is almost 20% of the instruction fetches. The next big bottleneck is the binding of variables on entry to a function, and this takes about 5.6% of the instruction fetches. Finally, to no ones surprise, the CONS routine takes about 5% of the instruction fetches. This is certainly high for fewer than thirty words of code, but it is not as bad as many people thought, given the complexity of the INTERLISP CONS algorithm.

Distribution of instruction fetch references for several programs

(Data from top 30 chunks, functionally distributed)

	Function Call	Entry	PDL search	type checking	CONS	
PARSEMEASURE	.073	.039	.455 (!)	.084	.024	
COMPILEMEASURE	.050	.130	.232	.126	.074	
DWIMIFY	.057	.092	.107	.133	.008	.071(IUB **Q)
WEST	.146	.117	.107	.104	.049	.017(IUB **Q)
COMNASAGRAMMAR	.140	.128	.214	.050	.055	
TASK1NLS	.100	.050	.157	.073	.061	
av.	.094	.093	.212	.082	.045	

Brief Program Descriptions:

PARSEMEASURE
June 1975 version of L. Bates' parser for the BBN speech understanding system, parsing a short sentence. Program not highly tuned.

COMPILE MEASURE
Compilation of 9 short and medium size functions from in-core definitions - compilation results stored in core and on a file. Program coded by systems personnel and carefully tuned.

DWIMIFY
Application of error correction function DWIMIFY to medium-size function containing CLISP expressions. Program carefully tuned and coded by system personnel.

WEST
Early version of a CAI program to teach arithmetic. Coded by non-systems personnel using a highly-modular, functionally decomposed style.

COMNASAGRAMMAR
Compiled version of ATN parser from the LUNAR natural language system. Code produced by grammar-compiler.

TASKINLS
LISP simulation of NLS system under control of a CAI lesson monitor and evaluator. Block-compiled system, moderately tuned.

Detailed description of the operation of the simulator and analyzer

We present below some fine details regarding the simulator and the way that page faulting data was analyzed. We hope that this information might be useful to anybody wishing to further analyze or interpret the data given in this report.

Since the simulator increases the CPU time needed to perform an operation by a factor of from 40 to 80, it is tempting to extract as much data as possible during a run of the simulator. This data can then be processed by any number of analysis programs to provide various characterizations of the operation of the program in executing the given job. However, there is a time/space tradeoff that arises that limits the amount of raw data that can be collected. Conceivably, one could write out on a file the entire sequence of instructions executed and the memory references made during the execution of a given user program. While this would give a complete record of the computational activity of the program, it is unfeasible for any but very short jobs - on a machine which normally executes 300,000 to 500,000 instructions per second, a few seconds of CPU time of the user job would produce enough data to fill an entire magnetic tape! Additionally, the i/o time needed to write out the volume of page reference data would be prohibitive.

Thus, the alternative tack was taken - certain measures of the memory referencing activity were abstracted during the simulation and then written out to be later analyzed. In all cases, parameters were accumulated for a quantum of 2048 memory references, and then the abstracted data were written out. Two distinct measures were made. The first measure was made in order to determine the page referencing activity of INTERLISP - this is the raw data used to determine properties of the INTERLISP "working set". To obtain this measure, the page number was obtained for every reference to memory (including those occurring during indirect reference chains). Two tables were kept, one containing the number of "instruction fetch" references to each page, and the other containing the number of read/write references. The reference counts were accumulated during a quantum (2048 total references) and then a record was written out indicating all pages which had been referenced during the quantum, and the number of read/write and fetch references actually made. In addition, the INTERLISP type table was saved for the given job, giving a record of the "type" of the page (i.e. whether it contained MACRO code, stack, lists, atom headers, compiled code and arrays, etc.) All measurements were made under conditions in which no garbage collections (which can cause page shuffling) would occur, so that the single type table was sufficient to record the characteristics of each page.

An added degree of subtlety had to be taken into account in recording page references, because of the "code swapping" or "compiled code overlay" facility of INTERLISP. INTERLISP maintains one (and potentially several) "lower forks" in which it stores compiled code. A segment of the basic 512k address space (generally 64 pages of 512 words each) is reserved as a "swapping buffer". By use of PMAP's this buffer is used to window sections of the lower fork(s) to run code, and therefore a reference to a "real" page in the swapping buffer is in actuality a reference to some "virtual" page in the lower fork. Thus, the potential address space of an INTERLISP program is not limited to the 512 pages directly addressable under TENEX - it can be indefinitely large, though in practice it is currently limited to 960 pages (1024 for two forks, minus 64 pages in the swapping buffer). It was decided to record the "virtual page" touched by each memory reference, so that we could tell which compiled code was being used, rather than simply what pages in the swapping buffer were being used to window compiled code. An added complication is that the assignment of pages in the lower fork to pages in the swapping buffer is dynamically variable, and so the simulator must make use of the INTERLISP swapper's tables to convert each reference to the swapping buffer to the current page reference in the lower fork.

In the section on Page Faulting vs. Working Set Size we indicated our use of a simplified page management algorithm (LRU) to replace the page management procedures actually used by TENEX. To make it possible to obtain page faulting behavior for different working set sizes with just a single pass over the data from the simulator, we make use of a related concept, the "distance string", rather than directly simulating the LRU algorithm.

Given a sequence of page references, the corresponding distance string is a sequence of numbers which gives, for each reference, the number of distinct pages which have been referenced since the last time the given page was referenced. Thus, given an LRU algorithm, for a fixed working set size all page references which have a distance string value greater than the working set size will cause faults, and all references with lower distances will not fault. This permits one to make a single run through the distance string file and compute the number of faults for any number of different working set sizes.

Most of our data comes in quantized sets of 2048 memory references, and thus we only know the time of reference of a page to within 2048 memory cycles. Because of this we must use an approximation to the distance string algorithm. The resulting analysis of our data is not exactly equivalent to the results of the simple LRU algorithm described above. For each page, we compute the number of _distinct_ pages which have been referenced since the last quantum in which the given page was referenced. We include in that count all pages referenced in the quantum when the given page was previously referenced which have not been referenced in the intervening quanta.

We have compared the 2048 memory reference quantum data with data taken with a quantum of 128 memory references (in which the average number of page references is slightly less than 10 per quantum). The graphs of a few of these runs are given below for comparison. The calculation of number of page faults for a given working set size is substantially the same (within a 2% range) for both quantum sizes, until the working set drops below 56 pages. This is an indication that the distance string values greater than 56 pages are quite accurate for the large quantum data, and since we are not extremely interested in the behavior of INTERLISP below about 75 pages (at which point it is already page-faulting almost every millisecond - a ridiculously high rate), the large quantum data is sufficient to characterize the paging performance of INTERLISP.

Some of the reasons why the large quantum approximation is likely to be fairly accurate for distance string values above 50 are:

a) On the average there are about 25 pages referenced in each quantum, and data indicates that 10 to 15 of those are referenced in almost every quantum. Thus, for distance string values greater than 50 - two quanta of references at least - the number of pages in the "previous reference quantum" which are not referenced in the intervening quanta is almost certain to be less than 10.

b) The data indicate that over 90% of the distance string values are below 60, so that for a page with distance string value over 60, chances are that the contribution from its "previous reference quantum" is less than 10% of the number of page references originally in that quantum, since the other 90% of those pages also occur in at least one of the intermediate quanta. Thus, the variation due to counting all of the remaining pages in the previous reference quantum is on the order of 10% of the number of pages in the quantum.

<u>Data from run of the dwimification, etc. example using a 128 memory reference quantum</u>

```
Example: DWCL128

1204224 Memory references in example


Allowed ¦ Page     :          Page faults per 20000 memory references
Working ¦ Faults   :
Set     ¦          :
        ¦          :
   248  ¦    240  :       *
   240  ¦    240  :       *
   232  ¦    241  :       *
   224  ¦    245  :       *
   216  ¦    248  :       *
   208  ¦    253  :       *
   200  ¦    260  :        *
   192  ¦    284  :        *
   184  ¦    314  :         *
   176  ¦    322  :          *
   168  ¦    334  :          *
   160  ¦    354  :           *
   152  ¦    375  :            *
   144  ¦    402  :            *
   136  ¦    420  :             *
   128  ¦    471  :               *
   120  ¦    569  :                  *
   112  ¦    619  :                    *
   104  ¦    712  :                       *
    96  ¦    829  :                           *
    88  ¦    980  :                                *
    80  ¦   1241  :                                        *
    72  ¦   1541  :                                                  *
    64  ¦   1916  :                                                  *
    56  ¦   2486  :                                                  *
    48  ¦   3431  :                                                  *
    40  ¦   5269  :                                                  *
                   ....................................................
                   :0         5        10        15        20       >>20
```

<u>Data from run of compilation using 128 memory reference quantum</u>

Example: SMALL128COM

2037760 Memory references in example

```
Allowed | Page     :          Page faults per 20000 memory references
Working | Faults   :
Set     |          :
        |          :
   240  |     236  :    *
   232  |     238  :    *
   224  |     241  :    *
   216  |     244  :    *
   208  |     251  :    *
   200  |     279  :     *
   192  |     290  :     *
   184  |     293  :     *
   176  |     302  :     *
   168  |     309  :     *
   160  |     322  :     *
   152  |     344  :      *
   144  |     368  :      *
   136  |     422  :       *
   128  |     521  :         *
   120  |     786  :               *
   112  |     898  :                 *
   104  |     963  :                  *
    96  |    1035  :                    *
    88  |    1132  :                      *
    80  |    1256  :                        *
    72  |    1425  :                           *
    64  |    1675  :                                 *
    56  |    2137  :                                          *
    48  |    2707  :                                                  *
    40  |    4139  :                                                  *
                   ....................................................
                   :0         5        10        15        20       >>20
```

Other page-fault versus working set curves

Example: COMPILEMEASURE

```
2928640 Memory references in example
Allowed ¦ Page      :          Page faults per 20000 memory references
Working ¦ Faults    :
Set     ¦           :
        ¦           :
   240  ¦    233    :  *
   232  ¦    234    :  *
   224  ¦    236    :  *
   216  ¦    240    :  *
   208  ¦    248    :  *
   200  ¦    285    :   *
   192  ¦    296    :   *
   184  ¦    300    :   *
   176  ¦    310    :   *
   168  ¦    321    :   *
   160  ¦    335    :    *
   152  ¦    355    :    *
   144  ¦    388    :    *
   136  ¦    474    :     *
   128  ¦    659    :        *
   120  ¦    945    :            *
   112  ¦   1067    :              *
   104  ¦   1156    :               *
    96  ¦   1232    :                *
    88  ¦   1342    :                 *
    80  ¦   1514    :                    *
    72  ¦   1731    :                       *
    64  ¦   2045    :                           *
    56  ¦   2544    :                                  *
    48  ¦   3706    :                                                  *
    40  ¦   6847    :                                                  *
                    ....................................................
                    :0        5        10        15        20       >>20
```

Example: EDIT/CLEANUP

8392704 Memory references in example

```
Allowed | Page     :          Page faults per 20000 memory references
Working | Faults   :
Set     |          :
        |          :
   320  |      317 : *
   312  |      317 : *
   304  |      319 : *
   296  |      339 : *
   288  |      346 : *
   280  |      354 : *
   272  |      366 : *
   264  |      377 : *
   256  |      396 : *
   248  |      415 : *
   240  |      460 : *
   232  |      485 : *
   224  |      523 :  *
   216  |      561 :  *
   208  |      602 :  *
   200  |      657 :  *
   192  |      723 :  *
   184  |      796 :   *
   176  |      875 :   *
   168  |      956 :    *
   160  |     1067 :    *
   152  |     1225 :     *
   144  |     1391 :      *
   136  |     1547 :      *
   128  |     1807 :        *
   120  |     2112 :         *
   112  |     2440 :           *
   104  |     2914 :             *
    96  |     3562 :                *
    88  |     4348 :                    *
    80  |     5530 :                         *
    72  |     7386 :                                  *
    64  |    10041 :                                               *
    56  |    14020 :                                                  *
    48  |    22092 :                                                  *
    40  |    53392 :                                                  *
                   ....................................................
                   :0         5        10        15        20       >>20
```

Example: NLSPARSE

473088 Memory references in example

```
Allowed ¦ Page      :          Page faults per 20000 memory references
Working ¦ Faults    :
Set     ¦           :
        ¦    205    :
   208  ¦    211    :                  *
   200  ¦    211    :                  *
   192  ¦    211    :                  *
   184  ¦    215    :                   *
   176  ¦    216    :                   *
   168  ¦    219    :                   *
   160  ¦    221    :                   *
   152  ¦    226    :                    *
   144  ¦    231    :                    *
   136  ¦    233    :                    *
   128  ¦    258    :                      *
   120  ¦    273    :                        *
   112  ¦    291    :                         *
   104  ¦    305    :                           *
    96  ¦    322    :                            *
    88  ¦    342    :                              *
    80  ¦    378    :                                 *
    72  ¦    458    :                                         *
    64  ¦    603    :                                                  *
    56  ¦    785    :                                                  *
    48  ¦   1200    :                                                  *
    40  ¦   2833    :                                                  *
                    :....................................................
                    :0         5        10        15        20      >>20
```

```
Example: PARSEMEASURE

4333568 Memory references in example


Allowed | Page
Working | Faults/
Set     | 20000 memory references
        |
   288  |    287   :  *
   280  |    289   :  *
   272  |    289   :  *
   264  |    289   :  *
   256  |    289   :  *
   248  |    289   :  *
   240  |    291   :  *
   232  |    295   :  *
   224  |    301   :  *
   216  |    312   :  *
   208  |    317   :  *
   200  |    327   :  *
   192  |    342   :  *
   184  |    365   :  *
   176  |    406   :   *
   168  |    453   :   *
   160  |    496   :    *
   152  |    545   :    *
   144  |    583   :    *
   136  |    625   :     *
   128  |    837   :       *
   120  |   1227   :          *
   112  |   1390   :            *
   104  |   1549   :             *
    96  |   1892   :                 *
    88  |   2173   :                   *
    80  |   2577   :                       *
    72  |   3329   :                              *
    64  |   5236   :                                                *
    56  |   7732   :                                                   *
    48  |  12129   :                                                   *
    40  |  29041   :                                                   *
                   .....................................................
                   :0         5        10         15        20      >>20
```

Official Distribution List
Contract N00014-75-C-1110

Defense Documentation Center
Cameron Station
Alexandria, Virginia 22314

TACTEC
Battelle Memorial Institute
505 King Avenue
Columbus, Ohio 43201

STOIAC
Battelle Memorial Institute
505 King Avenue
Columbus, Ohio 43201

Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217
Attn: Code 437

Office of Naval Research
Branch Office
495 Summer Street
Boston, Mass. 02210

Commander, Defense Contract
 Administration Services
Region - Boston
666 Summer Street
Boston, Mass. 02210

Director
Naval Research Laboratory
Washington, D.C. 20375
Attn: Code 2629

Director
Naval Research Laboratory
Washington, D.C. 20375
Attn: Code 2627

Assistant Chief for Technology
Office of Naval Research, Code 200
Arlington, Virginia 22217